Faculdade de Engenharia da Universidade do Porto

Rua Roberto Frias, sn, 4200-465 Porto, Portugal

Serviço Distribuido de Backups

Sistemas distribuídos

Relatório - TP1

Grupo: t4g08
Ana Cláudia Fonseca Santos - 200700742
Filipe Joaquim de Oliveira Reis Coelho - 201500072

9 de Abril de 2018

Introdução

O presente relatório serve para explicar ao pormenor os melhoramentos implementados nos protocolos desenvolvidos para a implementação do serviço de Backups, assim como a descrição da nossa solução para a concorrência. Para tal, foi necessário dividir este em 4 secções:

1 Backup

1.1 Especificação

Na especificação do protocolo foi pedido para dividir um ficheiro em *chunks* e enviar estes para a rede de *peers*. Cada um destes é responsável por o guardar e enviar a confirmação. Porém, da forma como o protocolo é descrito no enunciado, caso o número de *peers* fosse elevado e o grau de replicação desejado fosse baixo faria com que fosse ocupado mais espaço do que o necessário e assim esgotar o espaço de cada *peer* muito rapidamente.

1.2 Resolução

Cada vez que um *peer* recebe uma mensagem *PUTCHUNK* espera um tempo aleatório entre 0 e 400 ms. Durante esta espera são guardadas as mensagens do tipo *STORED* num *hashmap*. Após a espera verifica-se se o número de mensagens *STORED* referentes aquele chunk são inferiores ao grau de replicação desejado, se for guarda-se o chunk, senão é descartado.

O algoritmo utilizado:

```
Receive a PUTCHUNK message
wait time between 0 and 400ms;
menwhile, save in hashmap the number of STORED messages of that chunk

if(replication degree > number of stored's)
        ignore chunk
else
        store the chunk
        send STORED message
```

2 Restore

2.1 Especificação

No protocolo de *RESTORE* se os *chunks* forem grandes, e consequentemente as mensagens, não é adequado a utilização de um canal *multicast* para o seu envio, uma vez que apenas o *peer* que fez o pedido precisa de receber essa informação, evitando assim inundar a rede.

2.2 Resolução

De forma a resolver este problema, no construtor do *peer* é criado um *serverSocket* no endereço *localhost* e na porta peerId+6000 de forma a estabelecer ligação. Ao invocar a operação RESTORE o serverSocket é usado para criar um socket entre o peer receptor da CHUNK e o peer emissor. É realizada então a operação de envio do Chunk por TCP, na classe MessageUtils. São realizadas duas tentativas, com um delay entre envios de 400ms. Se o envio for bem sucedido é então recebido no peer que fez o pedido uma mensagem CHUNK serializada. Se não conseguir enviar por TCP nessas duas tentativas, faz *fallback* para o *multicast*, e prossegue com o envio por UDP (multicast).

No peer que fez o pedido foi implementada um *thread* que inicia um *listener* TCP que está conectado à respectiva *socket* à espera de receber dados, havendo na mesma um *thread* "que está atenta"à conexão *multicast*.

3 Delete

3.1 Especificação

Quando é feito o pedido de apagar um ficheiro, o *peer* que o recebe envia a mensagem *DELETE* para a rede *multicast*. Quando os restantes peers recebem esta mensagem apagam os fragmentos respectivos. No entanto se algum peer não estiver em execução os fragmentos são mantidos (uma vez que este não recebeu a mensagem) e assim, o espaço usado por estes nunca mais será recuperado.

3.2 Resolução

Quando é enviada uma mensagem *DELETE*, os peers além de apagarem os seus fragmentos também guardam o *fileId* numa *ConcurrentLinkedQueue*. Assim, quando um peer inicializa, este percorre os seus ficheiros e para cada um deles envia uma mensagem *GetDeleted* para os outros nós da rede para verificar se o ficheiro em questão se encontra nos *deleted files*. Se estiver, é enviada a mensagem *DELETE* e assim este apaga também os fragmentos do ficheiro do seu *filesystem*. Caso contrário estes são mantidos. Cada mensagem *GetDeleted* é enviada no máximo três vezes, com um *delay* se inicia de um segundo e que dobra a cada tentativa.

4 Concorrência

Para permitir a execução de vários protocolos em simultâneo é necessário lidar com concorrência. Quando um *peer* é inicializado cria uma *thread* para cada canal: canal de controlo (MC), canal de dados (MDB) e o canal de recuperação de dados (MDR).

No caso do canal de dados (fig.1) a *thread* principal está à espera de ler um datagrama e assim que este chega processa-o e cria um *thread* para executar o pedido correspondente. Para isso é usado o *ScheduledThreadPoolExecutor* da API do Java que cria uma *pool* de *Threads* para serem executadas num intervalo de tempo. Ainda neste exemplo, para lidar com mensagens do tipo *PUTCHUNK*, no canal MDB, é necessário uma estrutura de dados para guardar uma contagem das mensagens do tipo *STORED* (canal MC) ao mesmo tempo que aqui (MDB) se verifica se o grau de replicação desejado já foi alcançado (implica usar o método *get*). Deste modo é necessário que a estrutura de dados permita o uso seguro destas operações (get e put) de forma concorrente, e para tal foi usado o *ConcurrentHashMap*.

Foi necessário também lidar com concorrência no *peer* principal (aquele que comunica com o cliente) para verificar se os chunks da janela de transmissão anterior foram todos bem processados. Na thread que lida com o canal de controlo as mensagens do tipo STORED são adicionados à estrutura de dados enquanto que na thread que lida com o canal de dados a existência destas mensagens é verificada.

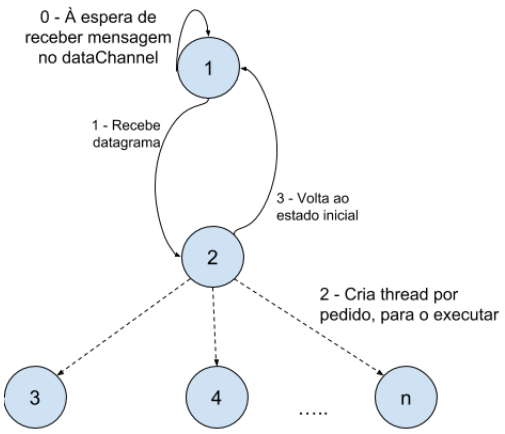

Figura 1: Maquina de estados da thread que lida com o MDB